

ML2 Final Project — Regression (Apartments Rent in Poland)

1. Project Overview

The objective of this project is to predict apartment rental prices in Poland using tree-based regression models.

Specifically, the following methods are considered:

1. Regression Tree,
2. Random Forest (Bagging),
3. Gradient Boosting (Boosting).

The modelling strategy follows the approaches presented during the ML2 lectures and focuses on reproducible experiments, clear and structured modelling pipelines, and a systematic comparison of tree-based methods.

Reproducibility

The experiments were conducted using the following environment:

- Python 3.12.7 (Anaconda distribution)
- scikit-learn 1.5.1
- pandas 2.2.2
- numpy 1.26.4

All results can be reproduced by running the notebook from top to bottom with the specified environment and the same random seed.

```
In [2]: import numpy as np
import pandas as pd

from sklearn.model_selection import train_test_split

RANDOM_STATE = 42
np.random.seed(RANDOM_STATE)

# Load data
DATA_PATH = "data/apartments_rent_pl_2024_04-2024_06.csv"
df = pd.read_csv(DATA_PATH)

print("Dataset shape:", df.shape)
display(df.head())
```

Dataset shape: (27542, 29)

		id	month	city	type	squareMeters	rooms	price
0		6ecaa957443f71ff320f9cf136ab416a	Apr-24	szczecin	blockOfFlats	28.60	1	270000.0000
1		9617848400b821533c7d4ceb75f9571c	Apr-24	szczecin	tenement	27.00	1	250000.0000
2		ffab4c61e84e7db6f8463cd54f063d74	Apr-24	szczecin	blockOfFlats	47.55	2	1000000.0000
3		03100b8d303bdabf71f32c1237e63e3f	Apr-24	szczecin	blockOfFlats	54.30	3	1000000.0000
4		30494a17380851da403d3c0848336566	Apr-24	szczecin	blockOfFlats	31.00	1	250000.0000

5 rows × 29 columns

```
In [4]: # Define target and features
TARGET_COL = "price"
X = df.drop(columns=[TARGET_COL])
y = df[TARGET_COL].astype(float)

print("Target summary:")
display(y.describe())
```

Target summary:

count	27542.00000
mean	3727.17319
std	2291.25990
min	412.00000
25%	2400.00000
50%	3000.00000
75%	4200.00000
max	19500.00000

Name: price, dtype: float64

```
In [6]: # Missing values (overview)
missing = df.isna().sum()
missing = missing[missing > 0].sort_values(ascending=False)

print("Number of columns with missing values:", missing.shape[0])
display(missing.head(20))
```

Number of columns with missing values: 14

condition	19841
buildingMaterial	11298
buildYear	7191
type	6052
floor	3278
hasElevator	1502
floorCount	473
collegeDistance	338
restaurantDistance	42
pharmacyDistance	27
clinicDistance	19
kindergartenDistance	9
postOfficeDistance	7
schoolDistance	4

dtype: int64

```
In [8]: # Split into train and test sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=RANDOM_STATE
)

print("Train shape:", X_train.shape, "| Test shape:", X_test.shape)
print("y_train mean:", y_train.mean(), "| y_test mean:", y_test.mean())
```

Train shape: (22033, 28) | Test shape: (5509, 28)
y_train mean: 3727.106839740389 | y_test mean: 3727.438555091668

2. Data preprocessing

Tree-based models in scikit-learn require:

- no missing values (no NaNs),
- all inputs to be numeric.

Therefore, we apply a minimal preprocessing pipeline:

- drop the identifier column (`id`),
- impute missing numeric values using the median,
- impute missing categorical values using the most frequent category,
- apply one-hot encoding to categorical variables.

This preprocessing is implemented using a scikit-learn `Pipeline` and `ColumnTransformer` to ensure reproducibility and prevent data leakage.

```
In [11]: from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import OneHotEncoder

# --- 1) Drop ID column if present ---
ID_COL = "id"
X_train_step2 = X_train.drop(columns=[ID_COL], errors="ignore")
X_test_step2 = X_test.drop(columns=[ID_COL], errors="ignore")

# --- 2) Identify feature types on training data only (prevents leakage) ---
categorical_cols = X_train_step2.select_dtypes(include=["object", "bool"]).columns
numeric_cols = X_train_step2.select_dtypes(include=[np.number]).columns.tolist()

print("Numeric features:", len(numeric_cols))
print("Categorical features:", len(categorical_cols))
print("Categorical columns:", categorical_cols)

# --- 3) Define preprocessing pipelines ---
numeric_transformer = Pipeline(steps=[
    ("imputer", SimpleImputer(strategy="median"))
])

categorical_transformer = Pipeline(steps=[
```

```

        ("imputer", SimpleImputer(strategy="most_frequent")),
        ("onehot", OneHotEncoder(handle_unknown="ignore"))
    ])

preprocessor = ColumnTransformer(
    transformers=[
        ("num", numeric_transformer, numeric_cols),
        ("cat", categorical_transformer, categorical_cols)
    ],
    remainder="drop"
)

# --- 4) Fit on train, transform train/test ---
X_train_processed = preprocessor.fit_transform(X_train_step2)
X_test_processed = preprocessor.transform(X_test_step2)

print("Processed train shape:", X_train_processed.shape)
print("Processed test shape:", X_test_processed.shape)

```

Numeric features: 16
Categorical features: 11
Categorical columns: ['month', 'city', 'type', 'ownership', 'buildingMaterial', 'condition', 'hasParkingSpace', 'hasBalcony', 'hasElevator', 'hasSecurity', 'hasStorageRoom']
Processed train shape: (22033, 52)
Processed test shape: (5509, 52)

In [13]:

```

# Basic sanity checks
# 1) Same number of features in train and test after preprocessing
assert X_train_processed.shape[1] == X_test_processed.shape[1], "Feature mismatch after preprocessing"

# 2) No missing values left
# For sparse matrices, checking NaN via sum is efficient
import numpy as np
train_has_nan = np.isnan(X_train_processed.data).any() if hasattr(X_train_processed,
test_has_nan = np.isnan(X_test_processed.data).any() if hasattr(X_test_processed,

print("NaN in processed train:", train_has_nan)
print("NaN in processed test :", test_has_nan)

```

NaN in processed train: False
NaN in processed test : False

3. Regression Tree

We start with a regression tree model (CART), which serves as a baseline for more advanced ensemble methods.

Regression trees are flexible but prone to overfitting.

Therefore, we apply cost-complexity pruning to control model complexity.

The pruning parameter α is selected using cross-validation.

```
In [18]: from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import mean_absolute_error, r2_score, root_mean_squared_error

def regression_metrics(y_true, y_pred):
    return {
        "MAE": mean_absolute_error(y_true, y_pred),
        "RMSE": root_mean_squared_error(y_true, y_pred),
        "R2": r2_score(y_true, y_pred)
    }
# Baseline (unpruned) tree
tree_baseline = DecisionTreeRegressor(random_state=RANDOM_STATE)
tree_baseline.fit(X_train_processed, y_train)

print("Baseline Regression Tree")
print("Train:", regression_metrics(y_train, tree_baseline.predict(X_train_processed)))
print("Test :", regression_metrics(y_test, tree_baseline.predict(X_test_processed)))

Baseline Regression Tree
Train: {'MAE': 1.405649102104419, 'RMSE': 49.89238255851453, 'R2': 0.999524292722293
2}
Test : {'MAE': 367.56416772554, 'RMSE': 780.3450327397613, 'R2': 0.8854832175467791}
```

```
In [20]: import numpy as np
import pandas as pd
from sklearn.model_selection import GridSearchCV
from sklearn.tree import DecisionTreeRegressor

# 1) Get meaningful alphas from pruning path
path = tree_baseline.cost_complexity_pruning_path(X_train_processed, y_train)
ccp_alphas = np.unique(path ccp_alphas)

# 2) Remove extremely large alpha (helps stability + plotting)
alpha_max = np.quantile(ccp_alphas, 0.99)
ccp_alphas = ccp_alphas[ccp_alphas <= alpha_max]

# 3) Sample a manageable number of candidates from the pruning path
max_candidates = 30
if len(ccp_alphas) > max_candidates:
    idx = np.linspace(0, len(ccp_alphas) - 1, max_candidates).astype(int)
    alpha_grid = ccp_alphas[idx]
else:
    alpha_grid = ccp_alphas

param_grid = {"ccp_alpha": alpha_grid}

search = GridSearchCV(
    estimator=DecisionTreeRegressor(random_state=RANDOM_STATE),
    param_grid=param_grid,
    scoring="neg_mean_squared_error",
    cv=3,
    n_jobs=-1
)

search.fit(X_train_processed, y_train)

best_alpha = float(search.best_params_["ccp_alpha"])
```

```
print("Best alpha (CV):", best_alpha)

results = pd.DataFrame(search.cv_results_)[["param_ccp_alpha", "mean_test_score", "cv_rmse"]]
results["cv_rmse"] = np.sqrt(-results["mean_test_score"])
results = results.sort_values("param_ccp_alpha")

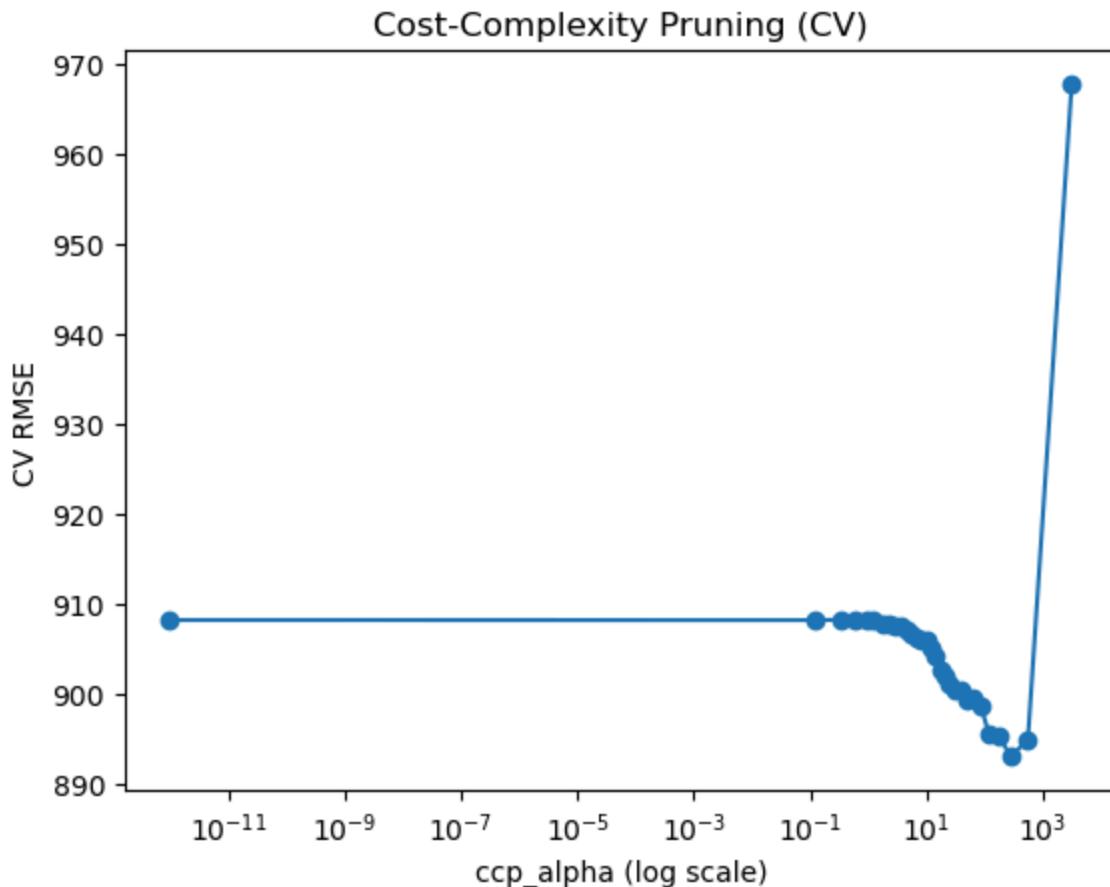
display(results)
```

Best alpha (CV): 276.95991968062253

	param_ccp_alpha	mean_test_score	rank_test_score	cv_rmse
0	0.000000	-824746.178517	27	908.155371
1	0.120047	-824722.657374	24	908.142421
2	0.348568	-824802.107499	29	908.186163
3	0.605153	-824727.855586	25	908.145283
4	0.915294	-824729.044728	26	908.145938
5	1.232252	-824794.428591	28	908.181936
6	1.728347	-824126.449067	23	907.814105
7	2.269323	-824041.349709	22	907.767233
8	2.889136	-823666.934962	21	907.560981
9	3.706561	-823521.614194	20	907.480917
10	4.543374	-822678.714206	19	907.016380
11	5.514456	-821861.725186	18	906.565897
12	6.821593	-821310.073359	17	906.261592
13	8.218477	-820767.213484	16	905.962038
14	9.795912	-820759.683861	15	905.957882
15	11.819392	-819056.047158	14	905.017153
16	14.434157	-817640.806137	13	904.234929
17	17.019925	-814865.034029	12	902.698750
18	20.418812	-813660.090099	11	902.031092
19	25.039924	-812034.478616	10	901.129557
20	31.320207	-810578.206652	8	900.321169
21	39.718850	-810802.021231	9	900.445457
22	49.630100	-808632.445242	6	899.239926
23	63.517038	-808998.418825	7	899.443394
24	85.593111	-807401.018640	5	898.554961
25	118.609964	-801782.490334	4	895.423079
26	176.346591	-801483.032075	3	895.255847
27	276.959920	-797449.016069	1	893.000009
28	548.457326	-800741.149188	2	894.841410
29	3106.644497	-936433.485235	30	967.694934

```
In [21]: import matplotlib.pyplot as plt
```

```
plt.figure()
plt.semilogx(results["param_ccp_alpha"].astype(float) + 1e-12, results["cv_rmse"],
plt.xlabel("ccp_alpha (log scale)")
plt.ylabel("CV RMSE")
plt.title("Cost-Complexity Pruning (CV)")
plt.show()
```



```
In [22]: # For Decision Trees, dense input can be faster than sparse in practice.
```

```
# Our feature dimension is small (52), so this is safe and efficient.
X_train_dense = X_train_processed.toarray() if hasattr(X_train_processed, "toarray")
X_test_dense = X_test_processed.toarray() if hasattr(X_test_processed, "toarray")

tree_pruned = DecisionTreeRegressor(
    random_state=RANDOM_STATE,
    ccp_alpha=float(best_alpha)
)

tree_pruned.fit(X_train_dense, y_train)

print("Pruned Regression Tree")
print("Train:", regression_metrics(y_train, tree_pruned.predict(X_train_dense)))
print("Test :", regression_metrics(y_test, tree_pruned.predict(X_test_dense)))
```

```
Pruned Regression Tree
Train: {'MAE': 386.51317259082145, 'RMSE': 515.0260945272086, 'R2': 0.94930912642160
28}
Test : {'MAE': 501.4288376410701, 'RMSE': 818.1865334278025, 'R2': 0.874107328057288
6}
```

Summary — Regression Tree

The unpruned regression tree shows severe overfitting, with near-perfect performance on the training set and substantially worse performance on the test set.

Using cost-complexity pruning and cross-validation, we select an optimal pruning parameter α that simplifies the model and reduces training overfitting. However, the test performance does not improve, indicating that a single regression tree has limited generalization ability for this problem.

This motivates the use of ensemble methods, such as Random Forests and Boosting, which are explored in the following steps.

4. Random Forest (Bagging)

A single regression tree suffers from high variance, as observed in the previous step. To reduce variance and improve generalization performance, we apply a Random Forest model, which is a bagging-based ensemble of regression trees.

Random Forests average predictions from multiple trees trained on bootstrap samples, thereby stabilizing predictions and reducing variance.

Selecting `max_depth` using Cross-Validation

To choose a reasonable tree depth for Random Forest, we evaluate a small set of candidate `max_depth` values using cross-validation on the training set. The best depth is selected based on the lowest cross-validated RMSE.

```
In [26]: import numpy as np
import pandas as pd
from sklearn.model_selection import KFold, cross_val_score
from sklearn.ensemble import RandomForestRegressor

# Candidate depths (keep it small and interpretable)
depth_candidates = [None, 10, 20, 30, 40, 50]

cv = KFold(n_splits=3, shuffle=True, random_state=RANDOM_STATE)

cv_results = []

for depth in depth_candidates:
    rf = RandomForestRegressor(
```

```

        n_estimators=200,           # 200 already stable in your results
        max_depth=depth,
        max_features="sqrt",
        random_state=RANDOM_STATE,
        n_jobs=-1
    )

    scores = cross_val_score(
        rf,
        X_train_processed,
        y_train,
        cv=cv,
        scoring="neg_root_mean_squared_error"
    )

    cv_rmse = -scores.mean()
    cv_std = scores.std()
    cv_results.append((depth, cv_rmse, cv_std))

cv_df = pd.DataFrame(cv_results, columns=["max_depth", "cv_rmse", "cv_rmse_std"])
best_depth = cv_df.loc[cv_df["cv_rmse"].idxmin(), "max_depth"]

print("Best max_depth (CV):", best_depth)
display(cv_df)

```

Best max_depth (CV): 40.0

	max_depth	cv_rmse	cv_rmse_std
0	NaN	704.244558	33.684914
1	10.0	901.912050	35.969099
2	20.0	706.080711	35.047224
3	30.0	704.920874	34.107977
4	40.0	704.204322	33.861383
5	50.0	704.244558	33.684914

In [27]:

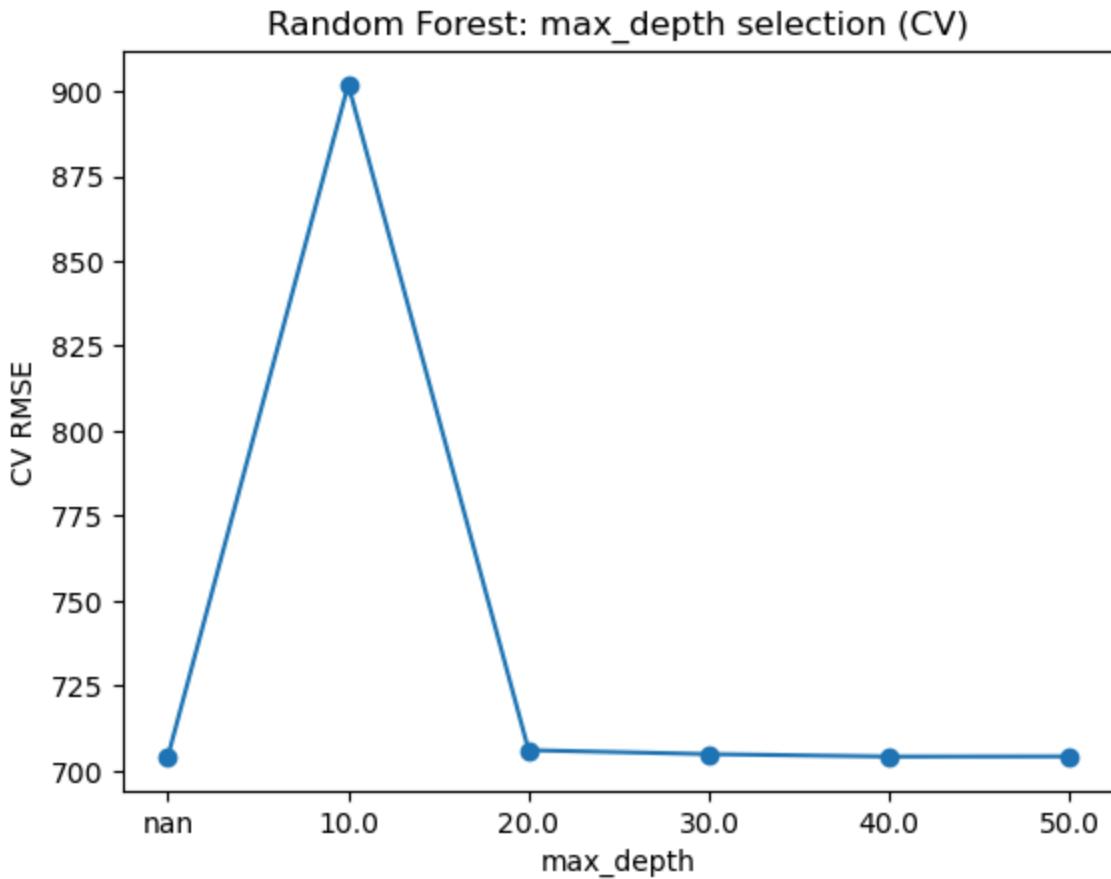
```

import matplotlib.pyplot as plt

plot_df = cv_df.copy()
plot_df["depth_label"] = plot_df["max_depth"].astype(str)

plt.figure()
plt.plot(plot_df["depth_label"], plot_df["cv_rmse"], marker="o")
plt.xlabel("max_depth")
plt.ylabel("CV RMSE")
plt.title("Random Forest: max_depth selection (CV)")
plt.show()

```



```
In [28]: from sklearn.ensemble import RandomForestRegressor

# Random Forest model (bagging)
rf_model = RandomForestRegressor(
    n_estimators=200,           # number of trees
    max_depth=None,            # allow trees to grow fully
    max_features="sqrt",       # standard choice for RF
    random_state=RANDOM_STATE,
    n_jobs=-1
)

rf_model.fit(X_train_processed, y_train)

# Predictions
y_train_pred_rf = rf_model.predict(X_train_processed)
y_test_pred_rf = rf_model.predict(X_test_processed)

print("Random Forest Regression")
print("Train:", regression_metrics(y_train, y_train_pred_rf))
print("Test :", regression_metrics(y_test, y_test_pred_rf))
```

```
Random Forest Regression
Train: {'MAE': 137.1576760718001, 'RMSE': 234.89019491901473, 'R2': 0.9894561046997885}
Test : {'MAE': 363.9045205710215, 'RMSE': 603.1631600661369, 'R2': 0.931582781722819}
```

```
In [29]: import pandas as pd
import matplotlib.pyplot as plt
```

```

comparison = pd.DataFrame({
    "Model": ["Regression Tree", "Pruned Tree", "Random Forest"],
    "Test_RMSE": [780.3, 818.2, 603.2]
})

plt.figure()
plt.bar(comparison["Model"], comparison["Test_RMSE"])
plt.ylabel("Test RMSE")
plt.title("Model Comparison on Test Set")
plt.show()

```



Summary — Random Forest (Bagging)

The Random Forest model substantially improves performance compared to a single regression tree. By averaging predictions from multiple trees trained on bootstrap samples, Random Forest effectively reduces variance.

Compared to both the baseline and pruned regression trees, the Random Forest achieves:

- a substantially lower test RMSE,
- a higher test R^2 ,
- and a noticeably smaller gap between training and test performance.

These results indicate that variance, rather than bias, is the main limitation of single-tree models in this dataset. By stabilizing predictions through bagging, Random Forest significantly improves generalization. However, while variance is effectively reduced, further

improvements may require addressing model bias, which is explored in the next step using boosting methods.

5. Boosting (Gradient Boosting)

While Random Forest significantly reduces variance by averaging multiple trees, it does not explicitly correct systematic prediction errors. To further improve performance by reducing bias, we apply Gradient Boosting, which builds trees sequentially, with each tree focusing on correcting the errors of the previous ensemble.

Hyperparameter Selection using Cross-Validation

Gradient Boosting performance is sensitive to hyperparameter choices. Therefore, we select key hyperparameters using cross-validation on the training set to obtain a more reliable estimate of generalization performance.

Consistent with the ML2 lecture material, I focus on a small and interpretable set of hyperparameters:

- `learning_rate`, controlling the contribution of each boosting stage,
- `max_depth`, determining the complexity of individual weak learners,
- `n_estimators`, specifying the number of boosting iterations.

The optimal configuration is selected based on the lowest cross-validated RMSE and subsequently evaluated on the held-out test set.

In [33]:

```
import numpy as np
import pandas as pd
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.model_selection import KFold, cross_val_score

# ---- CV setup (training set only) ----
cv = KFold(n_splits=3, shuffle=True, random_state=RANDOM_STATE)

# ---- A small, interpretable grid ----
param_grid = {
    "learning_rate": [0.2, 0.3],
    "max_depth": [4, 5],
    "n_estimators": [1000, 1200]
}

results = []

for lr in param_grid["learning_rate"]:
    for depth in param_grid["max_depth"]:
        for n_est in param_grid["n_estimators"]:
            gb = GradientBoostingRegressor(
                n_estimators=n_est,
                learning_rate=lr,
```

```

        max_depth=depth,
        random_state=RANDOM_STATE
    )

    scores = cross_val_score(
        gb,
        X_train_processed,
        y_train,
        cv=cv,
        scoring="neg_root_mean_squared_error"
    )
    cv_rmse_mean = -scores.mean()
    cv_rmse_std = scores.std()

    results.append({
        "learning_rate": lr,
        "max_depth": depth,
        "n_estimators": n_est,
        "cv_rmse_mean": cv_rmse_mean,
        "cv_rmse_std": cv_rmse_std
    })

cv_gb_df = pd.DataFrame(results).sort_values("cv_rmse_mean").reset_index(drop=True)

best_params = cv_gb_df.loc[0, ["learning_rate", "max_depth", "n_estimators"]].to_dict()
best_cv_rmse = cv_gb_df.loc[0, "cv_rmse_mean"]

print("Best GB params (CV):", best_params)
print("Best CV RMSE:", best_cv_rmse)

display(cv_gb_df.head(10))

```

Best GB params (CV): {'learning_rate': 0.2, 'max_depth': 5.0, 'n_estimators': 1200.0}

Best CV RMSE: 663.9144820097807

	learning_rate	max_depth	n_estimators	cv_rmse_mean	cv_rmse_std
0	0.2	5	1200	663.914482	25.172178
1	0.2	5	1000	665.139069	25.096575
2	0.2	4	1200	674.023160	34.054419
3	0.2	4	1000	676.740683	34.267232
4	0.3	5	1000	679.615552	33.120941
5	0.3	5	1200	679.955376	32.602099
6	0.3	4	1200	681.428454	28.595894
7	0.3	4	1000	683.407794	27.816991

In [34]: `from sklearn.ensemble import GradientBoostingRegressor`

```

gb_model = GradientBoostingRegressor(
    n_estimators=1200,

```

```

        learning_rate=0.2,
        max_depth=5,
        random_state=RANDOM_STATE
    )

gb_model.fit(X_train_processed, y_train)

# Predictions
y_train_pred_gb = gb_model.predict(X_train_processed)
y_test_pred_gb = gb_model.predict(X_test_processed)

print("Gradient Boosting Regression")
print("Train:", regression_metrics(y_train, y_train_pred_gb))
print("Test :", regression_metrics(y_test, y_test_pred_gb))

```

Gradient Boosting Regression
Train: {'MAE': 126.27665390961785, 'RMSE': 183.77149418173744, 'R2': 0.9935460202106724}
Test : {'MAE': 341.6776789139838, 'RMSE': 575.1729107147745, 'R2': 0.9377853526123187}

Summary — Gradient Boosting

Gradient Boosting improves predictive accuracy by building trees sequentially. Each new tree is fitted to the residual errors of the current ensemble, which helps reduce model bias.

Hyperparameters were selected using 3-fold cross-validation on the training set only. The best configuration was:

- learning_rate = 0.2
- max_depth = 5
- n_estimators = 1200

Using these parameters, the final model achieves on the held-out test set:

- Test RMSE ≈ 575.17
- Test R² ≈ 0.9378

Compared to Random Forest (bagging), Gradient Boosting yields better test performance, suggesting that explicitly correcting systematic prediction errors (bias reduction) provides additional gains beyond variance reduction alone.

6. Final Model Comparison

We compare all tree-based models using the same held-out test set. Lower RMSE indicates better predictive accuracy, while higher R² indicates a larger fraction of variance explained.

In [38]:

```

import pandas as pd
import matplotlib.pyplot as plt

```

```

# --- Helper to evaluate a fitted model on test set ---
def eval_on_test(model, X_test, y_test):
    y_pred = model.predict(X_test)
    m = regression_metrics(y_test, y_pred)
    return m["RMSE"], m["R2"], m["MAE"]

# --- Collect results ---
rmse_tree, r2_tree, mae_tree = eval_on_test(tree_baseline, X_test_processed, y_test)
rmse_pruned, r2_pruned, mae_pruned = eval_on_test(tree_pruned, X_test_processed, y_test)
rmse_rf, r2_rf, mae_rf = eval_on_test(rf_model, X_test_processed, y_test)
rmse_gb, r2_gb, mae_gb = eval_on_test(gb_model, X_test_processed, y_test)

comparison = pd.DataFrame([
    {"Model": "Regression Tree", "Test_RMSE": rmse_tree, "Test_R2": r2_tree, "Test_MAE": mae_tree},
    {"Model": "Pruned Tree", "Test_RMSE": rmse_pruned, "Test_R2": r2_pruned, "Test_MAE": mae_pruned},
    {"Model": "Random Forest", "Test_RMSE": rmse_rf, "Test_R2": r2_rf, "Test_MAE": mae_rf},
    {"Model": "Gradient Boosting", "Test_RMSE": rmse_gb, "Test_R2": r2_gb, "Test_MAE": mae_gb}
]).sort_values("Test_RMSE").reset_index(drop=True)

display(comparison)

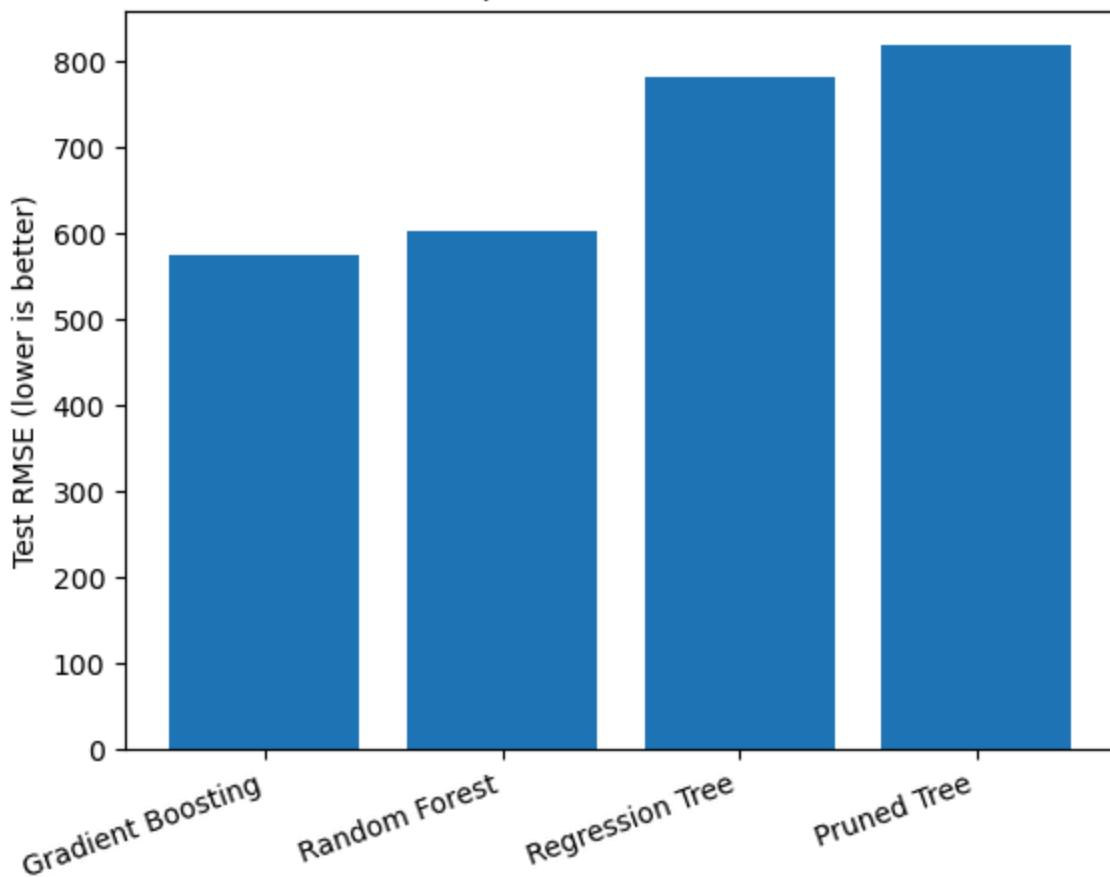
# --- Plot 1: Test RMSE ---
plt.figure()
plt.bar(comparison["Model"], comparison["Test_RMSE"])
plt.ylabel("Test RMSE (lower is better)")
plt.title("Model Comparison on Test Set – RMSE")
plt.xticks(rotation=20, ha="right")
plt.show()

# --- Plot 2: Test R² ---
plt.figure()
plt.bar(comparison["Model"], comparison["Test_R2"])
plt.ylabel("Test R² (higher is better)")
plt.title("Model Comparison on Test Set – R²")
plt.xticks(rotation=20, ha="right")
plt.show()

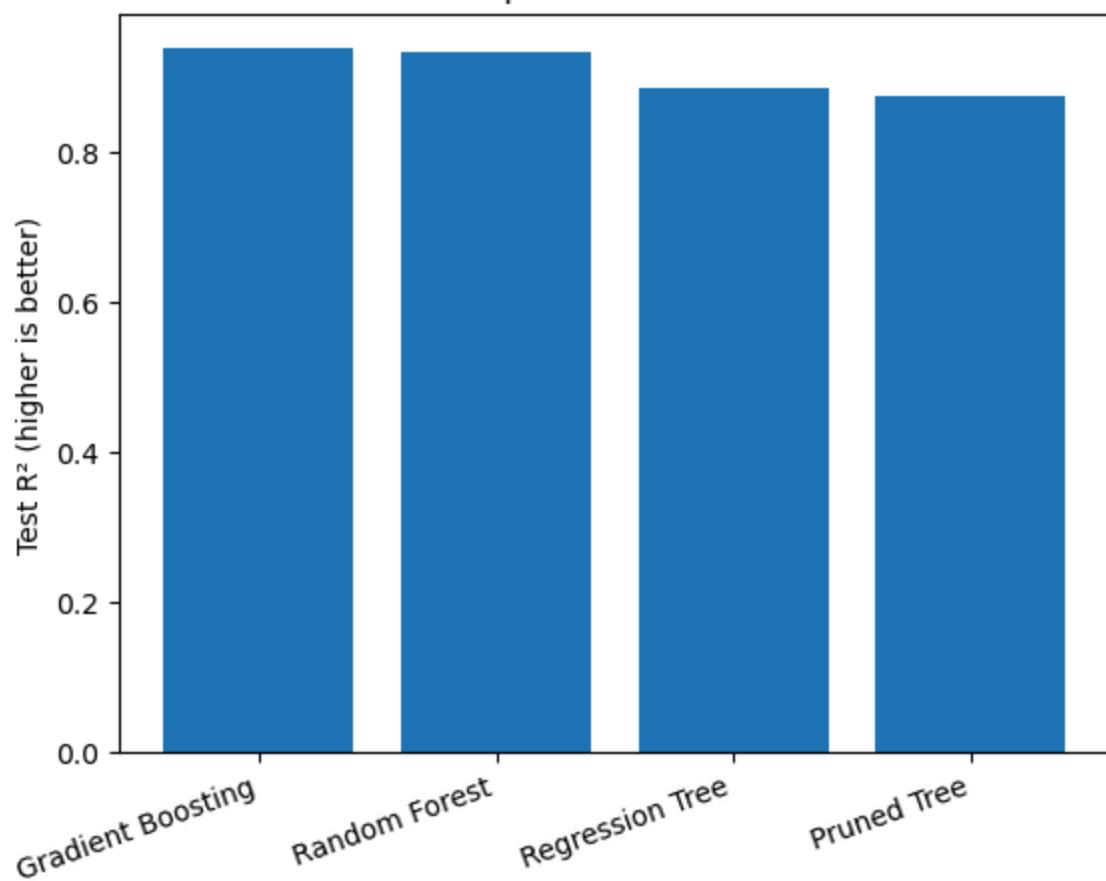
```

	Model	Test_RMSE	Test_R2	Test_MAE
0	Gradient Boosting	575.172911	0.937785	341.677679
1	Random Forest	603.163160	0.931583	363.904521
2	Regression Tree	780.345033	0.885483	367.564168
3	Pruned Tree	818.186533	0.874107	501.428838

Model Comparison on Test Set — RMSE



Model Comparison on Test Set — R^2



Summary — Final Comparison

On the held-out test set, Gradient Boosting achieves the best performance among the evaluated tree-based models (lowest RMSE and highest R²). This indicates that sequentially correcting previous errors (boosting) reduces bias and improves predictive accuracy compared to bagging (Random Forest) and single-tree baselines.

7. Error Analysis

Beyond aggregate metrics (RMSE/MAE/R²), we briefly analyze prediction errors on the held-out test set to understand where the final model performs less accurately. We focus on:

- residual patterns (systematic bias),
- error magnitude across different price ranges,
- examples of the largest errors.

```
In [61]: import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# --- Residuals & absolute errors ---
residuals = y_test - y_test_pred_gb
abs_errors = np.abs(residuals)

err_df = pd.DataFrame({
    "y_true": y_test,
    "y_pred": y_test_pred_gb,
    "residual": residuals,
    "abs_error": abs_errors
}).reset_index(drop=True)

# -----
# 1) Residual plot (predicted vs residual)
# -----
plt.figure()
plt.scatter(err_df["y_pred"], err_df["residual"], s=10)
plt.axhline(0)
plt.xlabel("Predicted price")
plt.ylabel("Residual (y_true - y_pred)")
plt.title("Residuals vs Predicted (Gradient Boosting) – Test Set")
plt.show()

# -----
# 2) Error by price range (quantiles of true price)
# -----
# Split the true prices into 4 equally-sized groups (quartiles)
err_df["price_bin"] = pd.qcut(err_df["y_true"], q=4, labels=["Q1 (low)", "Q2", "Q3"])

bin_summary = err_df.groupby("price_bin", observed=False).agg(
    mean_abs_error=("abs_error", "mean"),
```

```

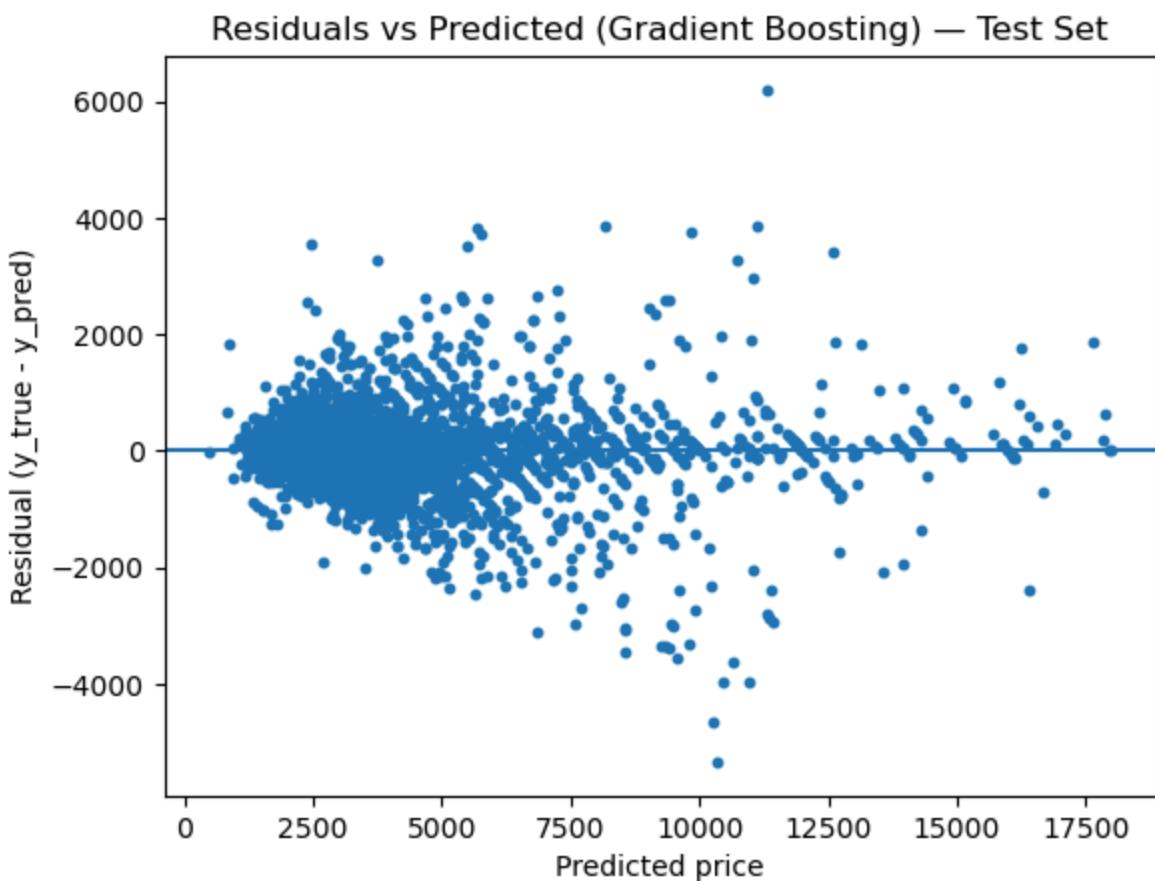
    median_abs_error=("abs_error", "median"),
    rmse=("residual", lambda x: np.sqrt(np.mean(np.square(x)))),
    count=("abs_error", "count")
).reset_index()

display(bin_summary)

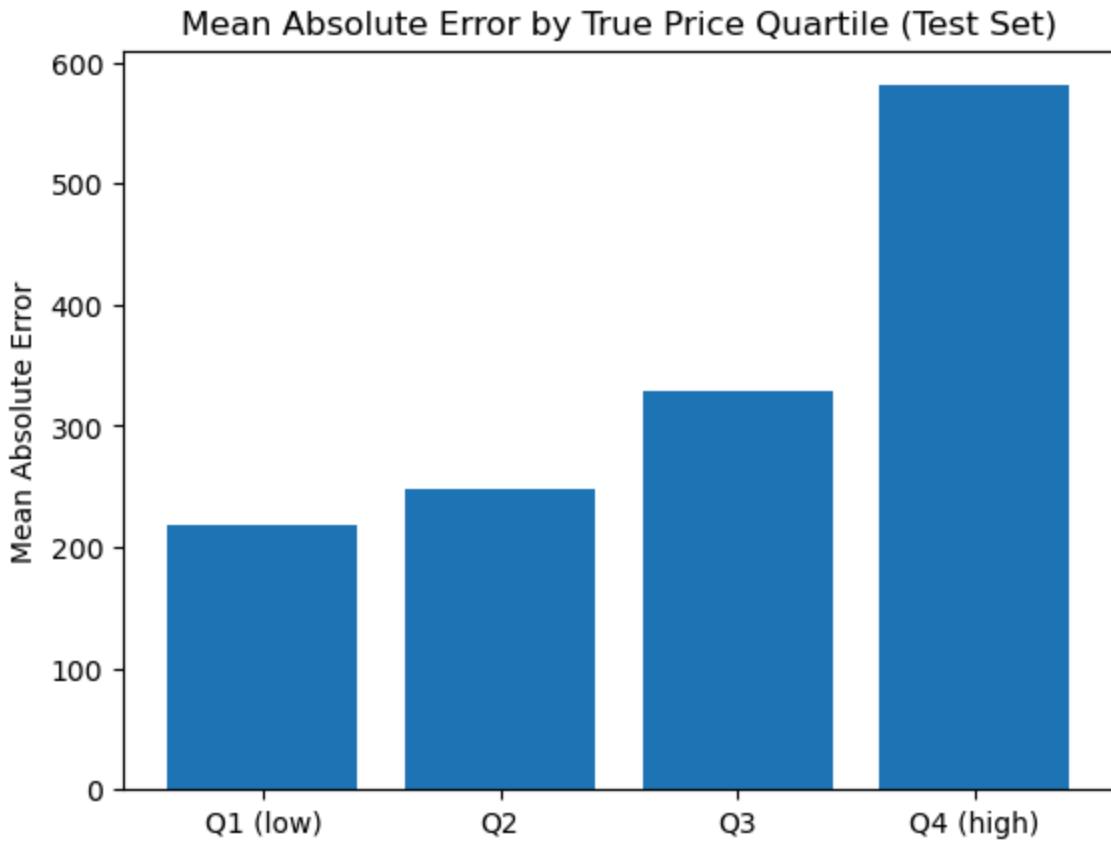
# Visualize mean absolute error by price bin
plt.figure()
plt.bar(bin_summary["price_bin"], bin_summary["mean_abs_error"])
plt.ylabel("Mean Absolute Error")
plt.title("Mean Absolute Error by True Price Quartile (Test Set)")
plt.show()

# -----
# 3) Inspect the largest errors (top 10)
# -----
top_errors = err_df.sort_values("abs_error", ascending=False).head(10)
display(top_errors)

```



	price_bin	mean_abs_error	median_abs_error	rmse	count
0	Q1 (low)	218.433100	154.359080	315.396828	1395
1	Q2	247.503612	162.994786	374.217117	1503
2	Q3	329.326939	227.140526	475.351003	1236
3	Q4 (high)	580.757916	296.432585	931.851842	1375



	y_true	y_pred	residual	abs_error	price_bin
811	17500.0	11305.348069	6194.651931	6194.651931	Q4 (high)
4003	5000.0	10334.350752	-5334.350752	5334.350752	Q4 (high)
4851	5600.0	10260.398196	-4660.398196	4660.398196	Q4 (high)
1495	7000.0	10980.545851	-3980.545851	3980.545851	Q4 (high)
4042	6500.0	10476.840346	-3976.840346	3976.840346	Q4 (high)
1154	15000.0	11132.276735	3867.723265	3867.723265	Q4 (high)
4417	12000.0	8151.094830	3848.905170	3848.905170	Q4 (high)
1364	9500.0	5690.397045	3809.602955	3809.602955	Q4 (high)
913	13600.0	9847.086511	3752.913489	3752.913489	Q4 (high)
258	9500.0	5761.901637	3738.098363	3738.098363	Q4 (high)

Summary - Error Analysis

The error summary by price quartiles shows larger errors for higher-priced apartments, which likely reflects higher heterogeneity (e.g., location, amenities, and listing quality) in the upper segment of the rental market.

8. Ethical Considerations

This model is trained on historical rental listings and therefore reflects existing market conditions and potential biases in the data (e.g., over-representation of certain cities, districts, or property types). If some regions or housing segments are under-sampled, predictions for those groups may be systematically less accurate, which can raise fairness concerns when the outputs are used in decision-making. In particular, higher-priced properties show larger prediction errors, likely due to higher heterogeneity in amenities, location, and listing quality in the upper market segment. Therefore, the model should not be used as the sole basis for pricing, screening, or automated decisions. Instead, it should be treated as a decision-support tool, combined with human review and domain knowledge, especially for out-of-distribution cases. When deployed, users should be informed of uncertainty and the model should be monitored for performance drift over time as market conditions change.

9. Final Conclusion

This project investigated apartment rental price prediction in Poland using tree-based regression models. Starting from a single regression tree, we progressively applied pruning, bagging (Random Forest), and boosting (Gradient Boosting) to improve predictive performance.

Empirical results show that ensemble methods substantially outperform single-tree models. Random Forest reduces variance and improves stability, while Gradient Boosting achieves the best overall performance by sequentially correcting residual errors, resulting in the lowest test RMSE and highest test R^2 .

Error analysis reveals that prediction uncertainty increases for higher-priced apartments, indicating greater variability in the upper tail of the price distribution. This highlights both the strengths and limitations of the model in practical applications.

Overall, the project demonstrates how different ensemble strategies affect bias–variance trade-offs and confirms the effectiveness of boosting methods for structured tabular regression problems.

References

- ML2 Lecture Materials: Regression Trees, Bagging, and Boosting.
- Apartments Prices in Poland dataset, Kaggle
(<https://www.kaggle.com/datasets/krzysztofjamroz/apartment-prices-in-poland>).